**Git Started**

The staging area is a file, generally contained in your Git directory, that stores information about what will go into your next commit. Its technical name in Git parlance is the “index”, but the phrase “staging area” works just as well.

Git comes with a tool called git config that lets you get and set configuration variables that control all aspects of how Git looks and operates. These variables can be stored in three different places

1. /etc/gitconfig file: Contains values applied to every user on the system and all their repositories. If you pass the option --system to git config, it reads and writes from this file specifically. (Because this is a system configuration file, you would need administrative or superuser privilege to make changes to it.)

2. ~/.gitconfig or ~/.config/git/config file: Values specific personally to you, the user. You can make Git read and write to this file specifically by passing the --global option, and this affects *all* of the repositories you work with on your system.

3. config file in the Git directory (that is, .git/config) of whatever repository you’re currently using: Specific to that single repository. You can force Git to read from and write to this file with the --local option, but that is in fact the default. (Unsurprisingly, you need to be located somewhere in a Git repository for this option to work properly.)

Each level overrides values in the previous level

On Windows systems, Git looks for the .gitconfig file in the $HOME directory (C:Users\$USER for most people). It also still looks for /etc/gitconfig, although it’s relative to the MSys root, which is wherever you decide to install Git on your Windows system when you run the installer. If you are using version 2.x or later of Git for Windows, there is also a system-level config file at C:\Documents and Settings\All Users\Application Data\Git\config on Windows XP, and in C:\ProgramData\Git\config on Windows Vista and newer. This config file can only be changed by git config -f <file> as an admin.

**Your Identity**

The first thing you should do when you install Git is to set your user name and email address. This is important because every Git commit uses this information, and it’s immutably baked into the commits you start creating:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

**Your Editor**

Now that your identity is set up, you can configure the default text editor that will be used when Git needs you to type in a message. If not configured, Git uses your system’s default editor.If you want to use a different text editor, such as Emacs, you can do the following:

$ git config --global core.editor emacs

If you want to check your configuration settings, you can use the git config --list command to list all the settings Git can find at that point:

You can also check what Git thinks a specific key’s value is by typing git config <key>:

$ git config user.name

Since Git might read the same configuration variable value from more than one file, it’s possible that you have an unexpected value for one of these values and you don’t know why. In cases like that, you can query Git as to the *origin* for that value, and it will tell you which configuration file had the final say in setting that

value:

$ git config --show-origin rerere.autoUpdate

file:/home/johndoe/.gitconfig false

If you ever need help while using Git, there are two equivalent ways to get the comprehensive manual page (manpage) help for any of the Git commands:

$ git help <verb>

In addition, if you don’t need the full-blown manpage help, but just need a quick refresher on the available options for a Git command, you can ask for the more concise “help” output with the -h or --help options, as in:

$ git add –h

**Git Basics**

**Short Status**

While the git status output is pretty comprehensive, it’s also quite wordy. Git also has a short status flag so you can see your changes in a more compact way. If you run git status -s or git status --short you get a far more simplified output from the command:

$ git status –s

New files that aren’t tracked have a ?? next to them, new files that have been added to the staging area have an A, modified files have an M and so on. There are two columns to the output - the lefthand column indicates the status of the staging area and the right-hand column indicates the status of the working tree. So for example in that output, the README file is modified in the working directory but not yet staged, while the lib/simplegit.rb file is modified and staged. The Rakefile was modified, staged and then modified again, so there are changes to it that are both staged and unstaged.

**Ignoring Files**

Often, you’ll have a class of files that you don’t want Git to automatically add or even show you as being untracked. These are generally automatically generated files such as log files or files produced by your build system. In such cases, you can create a file listing patterns to match them named .gitignore. Here is an example .gitignore file:

$ cat .gitignore

\*.[oa]

\*~

The first line tells Git to ignore any files ending in “.o” or “.a” — object and archive files that may be the product of building your code. The second line tells Git to ignore all files whose names end with a tilde (~), which is used by many text editors such as Emacs to mark temporary files. You may also include a log, tmp, or pid directory; automatically generated documentation; and so on. Setting up a .gitignore file for your new repository before you get going is generally a good idea so you don’t accidentally commit files that you really don’t want in your Git repository. The rules for the patterns you can put in the .gitignore file are as follows:

* Blank lines or lines starting with # are ignored.
* Standard glob patterns work, and will be applied recursively throughout the entire working tree.
* You can start patterns with a forward slash (/) to avoid recursivity.
* You can end patterns with a forward slash (/) to specify a directory.
* You can negate a pattern by starting it with an exclamation point (!).

Glob patterns are like simplified regular expressions that shells use. An asterisk (\*) matches zero or more characters; [abc] matches any character inside the brackets (in this case a, b, or c); a question mark (?) matches a single character; and brackets enclosing characters separated by a hyphen ([0-9]) matches any character between them (in this case 0 through 9). You can also use two asterisks to match nested directories; a/\*\*/z would match a/z, a/b/z, a/b/c/z, and so on. Here is another example .gitignore file:

# ignore all .a files

\*.a

# but do track lib.a, even though you're ignoring .a files above

!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO

/TODO

# ignore all files in any directory named build

build/

# ignore doc/notes.txt, but not doc/server/arch.txt

doc/\*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories

doc/\*\*/\*.pdf

**Viewing Your Staged and Unstaged Changes**

To see what you’ve changed but not yet staged, type git diff with no other arguments. That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use git diff --staged. This command compares your staged changes to your last commit:

It’s important to note that git diff by itself doesn’t show all changes made since your last commit — only changes that are still unstaged. If you’ve staged all of your changes, git diff will give you no output.

Now you can use git diff to see what is still unstaged and git diff --cached to see what you’ve staged so far (--staged and --cached are synonyms):

*Git Diff in an External Tool*

We will continue to use the git diff command in various ways throughout the rest of the book. There is another way to look at these diffs if you prefer a graphical or external diff viewing program instead. If you run git difftool instead of git diff, you can view any of these diffs in software like emerge, vimdiff and many more (including commercial products). Run git difftool --tool-help to see what is available on your system

If you want to skip the staging area, Git provides a simple shortcut. Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit,

$ git commit -am 'added new benchmarks'

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The git rm command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around.

The next time you commit, the file will be gone and no longer tracked. If you modified the file and added it to the staging area already, you must force the removal with the -f option. This is a safety feature to prevent accidental removal of data that hasn’t yet been recorded in a snapshot and that can’t be recovered from Git.

Another useful thing you may want to do is to keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore. This is particularly useful if you forgot to add something to your .gitignore file and accidentally staged it, like a large log file or a bunch of .a compiled files. To do this, use the --cached option:

$ git rm --cached README

That means you can do things such as:

$ git rm log/\*.log

$ git rm /\*~

**Moving Files**

Unlike many other VCS systems, Git doesn’t explicitly track file movement. If you rename a file in Git, no metadata is stored in Git that tells it you renamed the file. However, Git is pretty smart about figuring that out after the fact — we’ll deal with detecting file movement a bit later. Thus it’s a bit confusing that Git has a mv command. If you want to rename a file in Git, you can run something like:

$ git mv file\_from file\_to

However, this is equivalent to running something like this:

$ mv README.md README

$ git rm README.md

$ git add README

Git figures out that it’s a rename implicitly, so it doesn’t matter if you rename a file that way or with the mv command. The only real difference is that git mv is one command instead of three — it’s a convenience function. More importantly, you can use any tool you like to rename a file, and address

the add/rm later, before you commit.

**Viewing the Commit History**

The most basic and powerful tool to do this is the git log command.

One of the more helpful options is -p or --patch, which shows the difference (the *patch* output) introduced in each commit. You can also limit the number of log entries displayed, such as using -2 to show only the last two entries.

$ git log -p -2

if you want to see some abbreviated stats for each commit, you can use the --stat option

As you can see, the --stat option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end.

Another really useful option is --pretty. This option changes the log output to formats other than the default. A few prebuilt options are available for you to use. The oneline option prints each commit on a single line, which is useful if you’re looking at a lot of commits. In addition, the short, full, and fuller options show the output in roughly the same format but with less or more information, respectively:

$ git log --pretty=oneline

The most interesting option is format, which allows you to specify your own log output format. This is especially useful when you’re generating output for machine parsing — because you specify the format explicitly, you know it won’t change with updates to Git:

$ git log --pretty=format:"%h - %an, %ar : %s"

ca82a6d - Scott Chacon, 6 years ago : changed the version number

085bb3b - Scott Chacon, 6 years ago : removed unnecess

**Option Description of Output**

%H Commit hash

%h Abbreviated commit hash

%T Tree hash

%t Abbreviated tree hash

%P Parent hashes

%p Abbreviated parent hashes

%an Author name

%ae Author email

%ad Author date (format respects the --date=option)

%ar Author date, relative

%cn Committer name

%ce Committer email

%cd Committer date

%cr Committer date, relative

%s Subject

The oneline and format options are particularly useful with another log option called --graph. This option adds a nice little ASCII graph showing your branch and merge history:

*Table 2. Common options to* git log

**Option Description**

-p Show the patch introduced with each commit.

--stat Show statistics for files modified in each commit.

--shortstat Display only the changed/insertions/deletions line from the --stat command.

--name-only Show the list of files modified after the commit information.

--name-status Show the list of files affected with added/modified/deleted information as well.

--abbrev-commit Show only the first few characters of the SHA-1 checksum instead of all 40.

--relative-date Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.

--graph Display an ASCII graph of the branch and merge history beside the log output.

--pretty Show commits in an alternate format. Options include oneline, short, full,fuller, and format (where you specify your own format).

--oneline Shorthand for --pretty=oneline --abbrev-commit used together.

**Limiting Log Output**

In addition to output-formatting options, git log takes a number of useful limiting options

* -<n>, where n is any integer to show the last n commits
* --since and --until the time-limiting options.

$ git log --since=2.weeks

This command works with lots of formats — you can specify a specific date like "2008-01-15", or a relative date such as "2 years 1 day 3 minutes ago"

* The --author option allows you to filter on a specific author,
* the --grep option lets you search for keywords in the commit messages

Another really helpful filter is the -S option (colloquially referred to as Git’s “pickaxe” option), which takes a string and shows only those commits that changed the number of occurrences of that string.

$ git log -S FindNextStructure

The last really useful option to pass to git log as a filter is a path. If you specify a directory or file name, you can limit the log output to commits that introduced a change to those files. This is always the last option and is generally preceded by double dashes (--) to separate the paths from the options.

**Option Description**

-<n> Show only the last n commits

--since, --after Limit the commits to those made after the specified date.

--until, --before Limit the commits to those made before the specified date.

--author Only show commits in which the author entry matches the specified string.

--committer Only show commits in which the committer entry matches the specified string.

--grep Only show commits with a commit message containing the string

-S Only show commits adding or removing code matching the string

$ git log --pretty="%h - %s" --author='Junio C Hamano' --since="2008-10-01"

--before="2008-11-01" --no-merges -- testData/

To prevent the display of merge commits cluttering up your log history, simply add the log option --no-merges

**Undoing Things**

If you want to redo that commit, make the additional changes you forgot, stage them, and commit again using the --amend option:

$ git commit –amend

**Unstaging a Staged File**

git reset HEAD <file> to unstagea file

**Unmodifying a Modified File**

$ git checkout -- CONTRIBUTING.md

It’s important to understand that git checkout -- <file> is a dangerous command. Any changes you made to that file are gone — Git just copied another file over it. Don’t ever use this command unless you absolutely know that you don’t want the file.

Remember, anything that is *committed* in Git can almost always be recovered. Even commits that were on branches that were deleted or commits that were overwritten with an --amend commit can be recovered (see Data Recovery for data recovery). However, anything you lose that was never committed is likely never to be seen again.

**Working with Remotes**

You can also specify -v, which shows you the URLs that Git has stored for the shortname to be used when reading and writing to that remote:

**Adding Remote Repositories**

To add a new remote Git repository as a shortname you can reference easily, run

$ git remote add <shortname> <url>:

**Fetching and Pulling from Your Remotes**

As you just saw, to get data from your remote projects, you can run:

$ git fetch <remote>

It’s important to note that the git fetch command only downloads the data to your local repository —

git pull command to automatically fetch and then merge that remote branch into your current branch

Running git pull generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

**Pushing to Your Remotes**

The command for this is simple: git push <remote> <branch>. If you want to push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically), then you can run this to push any commits you’ve done back up to the server:

$ git push origin master

**Inspecting a Remote**

If you want to see more information about a particular remote, you can use the git remote show <remote> command

$ git remote show origin

This command shows which branch is automatically pushed to when you run git push while on certain branches. It also shows you which remote branches on the server you don’t yet have, which remote branches you have that have been removed from the server, and multiple local branches that are able to merge automatically with their remote-tracking branch when you run git pull.

**Renaming and Removing Remotes**

For instance, if you want to rename pb to paul, you can do so with git remote rename

If you want to remove a remote for some reason — you’ve moved the server or are no longer using a particular mirror, or perhaps a contributor isn’t contributing anymore — you can either use git remote remove or git remote rm:

**Tagging**

**Listing Your Tags**

Listing the available tags in Git is straightforward. Just type git tag (with optional -l or --list):

$ git tag

You can also search for tags that match a particular pattern

$ git tag -l "v1.8.5\*"

If you want just the entire list of tags, running the command git tag implicitly assumes you want a listing and provides one; the use of -l or --list in this case is optional. If, however, you’re supplying a wildcard pattern to match tag names, the use of –l or --list is mandatory.

**Creating Tags**

Git supports two types of tags: *lightweight* and *annotated*.

A lightweight tag is very much like a branch that doesn’t change — it’s just a pointer to a specific commit.

Annotated tags, however, are stored as full objects in the Git database. The easiest way is to specify -a when you run the tag command:

$ git tag -a v1.4 -m "my version 1.4"

**Git distribue**

lancez donc git log --no-merges dessus pour voir a quoi ressemble un historique de *commits* avec des messages bien formates.

La difference principale reste que les fusions ont lieu du cote client plutot que

sur le serveur au moment de valider

Il est tres important de comprendre ceci si vous avez deja utilise Subversion, parce qu’il faut remarquer que les deux developpeurs n’ont pas modifie le meme fichier. Quand des fichiers differents ont été modifies, Subversion realise cette fusion automatiquement sur le serveur alors que Git necessite une fusion des modifications locale. John doit recuperer les modifications de Jessica et les fusionner avant d’etre autorise a pousser :

$ git log --no-merges issue54..origin/master

La syntaxe issue54..origin/master est un filtre du journal qui ordonne a Git de ne montrer que la liste des *commits* qui sont sur la seconde branche (dans ce cas origin/master) et qui ne sont pas sur la premiere (dans ce cas prob54). Nous aborderons cette syntaxe en detail dans Plages de *commits.*

Mais il y a un petit probleme — elle doit pousser son travail fusionne dans sa branche fonctionB sur la branche fonctionBee du serveur. Elle peut le faire en specifiant la branche locale suivie de deux points (:) suivi de la branche distante a la commande git push :

$ git push -u origin fonctionB:fonctionBee

Notez l’option -u. C’est un raccourci pour --set-upstream, qui configure les branches pour faciliter les poussees et les tirages plus tard.

Vous demarrez une nouvelle branche a partir de la branche origin/master

courante, y collez les modifications de fonctionB en resolvant les conflits, changez l’implementation et poussez le tout en tant que nouvelle branche

$ git checkout -b fonctionBv2 origin/master

$ git merge --no-commit --squash fonctionB

# (changement d'implementation)

$ git commit

$ git push macopie fonctionBv2

.

gitk is a graphical history viewer. Think of it like a powerful GUI shell over git log and git grep

Gitk is easiest to invoke from the command-line. Just cd into a Git repository, and type:

$ gitk [git log options]

Gitk accepts many command-line options, most of which are passed through to the underlying git log action. Probably one of the most useful is the --all flag, which tells gitk to show commits reachable from *any* ref, not just HEAD.

git-gui, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

$ git gui

Pro Git (Vf)

$ git log –abrev-commit –pretty=oneline

Pour connaitre l’empreinte SHA sur laquelle pointe une branche

$ git rev-parse topic1

Git maintient en arriere-plan un historique des references ou sont passes HEAD et vos branches sur les derniers mois — ceci s’appelle le *reflog*. Vous pouvez le consulter avec la commande git reflog :

Si vous souhaitez consulter le n-ieme antecedent de votre HEAD, vous pouvez utiliser la reference @{n} du reflog, 5 dans cet exemple :

$ git show HEAD@{5}

Pour consulter le reflog au format git log, executez: git log -g. Veuillez noter que le reflog ne stocke que des informations locales, c’est un historique de ce que

vous avez fait dans votre depot

Alors, vous pouvez consulter le *commit* precedent en specifiant HEAD^, ce qui signifie ≪ le parent de HEAD ≫:

Vous pouvez egalement specifier un nombre apres ^ — par exemple, d921970^2 signifie ≪ le second parent de d921970 ≫. Cette syntaxe ne sert que pour les *commits* de fusion, qui ont plus d’un parent. Le premier parent est la branche depuis laquelle vous avez fusionne, et le second est le *commit* de la branche que vous avez fusionnee :

Une autre solution courante pour specifier une reference ancetre est le ~. Il fait egalement reference au premier parent, donc HEAD~ et HEAD^ sont equivalents. La difference apparait si vous specifiez un nombre. HEAD~2 signifie ≪ le premier parent du premier parent ≫, ou bien ≪ le grandparent ≫ ; on remonte les premiers parents autant de fois que demande.

HEAD~3 peut aussi s’ecrire HEAD^^^.

Vous pouvez egalement combiner ces syntaxes HEAD~3^2,

**Plages de** *commits*

**Double point**

Si vous voulez savoir ce qui n’a pas encore ete fusionne sur votre branche master depuis votre branche experiment, vous pouvez demander a Git de vous montrer un journal des *commits* avec master..experiment

$ git log origin/master..HEAD

Cette commande vous affiche tous les *commits* de votre branche courante qui ne sont pas sur la branche master du depot distant origin.

Vous pouvez egalement laisser tomber une borne de la syntaxe pour faire comprendre a Git que vous parlez de HEAD. Par exemple, vous pouvez obtenir les memes resultats que precedemment en tapant git log origin/master.. — Git utilise HEAD si une des bornes est manquante.

**Emplacements multiples**

Git vous permet cela avec ^ ou --not en prefixe de toute reference de laquelle vous ne souhaitez pas voir les *commits*. Les 3 commandes ci-apres sont equivalentes :

$ git log refA..refB

$ git log ^refA refB

$ git log refB --not refA

C’est utile car cela vous permet de specifier plus de 2 references dans votre requete, ce que vous ne pouvez accomplir avec la syntaxe double-point. Par exemple, si vous souhaitez voir les *commits* qui sont accessibles depuis refA et refB mais pas depuis refC, vous pouvez taper ces 2 commandes :

$ git log refA refB ^refC

$ git log refA refB --not refC

**Triple point**

La derniere syntaxe majeure de selection de plage de *commits* est la syntaxe triple-point qui specifie tous les *commits* accessibles par l’une des deux references, exclusivement

$ git log master...experiment

Une option courante a utiliser avec la commande log dans ce cas est --left-right qui vous montre la borne de la plage a laquelle ce *commit* appartient. Cela rend les donnees plus utiles :

$ git log --left-right master...experiment

Mode patch

$ git add –p

$ git reset –p

$ git checkout –p

$ git stash save -p

Remiser votre travail

$ git stash save

$ git stash list

$ git stash apply

vous voulez appliquer une remise plus ancienne, vous pouvez la specifier en la nommant, comme ceci : git stash apply stash@{2}. Si vous ne specifiez pas une remise, Git presume que vous voulez la remise la plus recente et essaye de l’appliquer

Par defaut, les modifications de vos fichiers sont re-appliquees, mais pas les indexations. Pour cela, vous devez executer la commande git stash apply avec l’option --index pour demander a Git d’essayer de re-appliquer les modifications de votre index.

Pour la supprimer, vous pouvez executer git stash drop avec le nom de la remise a supprimer

Vous pouvez egalement executer git stash pop pour appliquer et supprimer immediatement la remise de votre pile.

Il existe des variantes de remisages qui peuvent s’averer utiles.La premiere option

est l’option --keep-index de la commande stash save. Elle indique a Git de ne pas remiser ce qui aurait ete deja indexe au moyen de la commande git add.

Une autre option utile de stash est la possibilite de remiser les fichiers non suivis aussi bien que les fichiers suivis. Par defaut, git stash ne sauve que les fichiers qui sont deja suivis ou indexes. Si vous specifiez l’option --include-untracked ou -u, Git remisera aussi les fichiers non-suivis du repertoire de travail

Une option plus securisee consiste a lancer git stash --all pour tout sauvegarder dans une remise.

**Creer une branche depuis une remise**

vous pouvez executer git stash branch qui creera une nouvelle branche a votre place, recuperant le *commit* ou vous etiez lorsque vous avez cree la remise, re-appliquera votre travail dedans, et supprimera finalement votre remise si cela a reussi

$ git stash branch testchanges

**Nettoyer son repertoire de travail**

Pour supprimer tous les fichiers non-suivis, vous pouvez lancer git clean -f -d, qui effacera aussi tout sous-repertoire vide. L’option -f signifie ≪ force ≫, soit ≪ fais-le reellement ≫.

$ git clean -d -n

Si vous souhaitez visualiser ce qui serait fait, vous pouvez lancer la commande avec l’option -n qui signifie ≪ fais-le a blanc et montre-moi ce qui *serait* supprime ≫.

$ git clean -d –n

Par defaut, la commande git clean ne va supprimer que les fichiers non-suivis qui ne sont pas ignores. Tout fichier qui correspond a un motif de votre fichier .gitignore ou tout autre fichier similaire ne sera pas supprime. Si vous souhaitez supprimer aussi ces fichiers, comme par exemple les fichiers .o genere par un compilateur pour faire une compilation totale, vous pouvez ajouter l’option -x a la commande de nettoyage

Si vous ne savez pas ce que la commande git clean va effectivement supprimer, lancez-la une premiere fois avec -n par securite avant de transformer le -n en -f et nettoyer definitivement. Un autre choix pour s’assurer de ce qui va etre efface consiste a lancer la commande avec l’option –i ou --interactive.

La commande sera lancee en mode interactif.

$ git clean -x –i

**Recherche**

Git fournit quelques outils permettant rapidement de rechercher dans le code et les *commits* stockes dans votre base de données

**Git grep**

grep permet de rechercher facilement une chaine de caracteres ou une expression reguliere dans une arborescence validee ou dans le repertoire de travail. Pour tous les exemples qui suivent, nous allons utiliser le depot de Git lui-meme. Par defaut, git grep recherche dans le repertoire de travail. Vous pouvez passer l’option -n pour

afficher les numeros des lignes des correspondances.

$ git grep -n gmtime\_r

La commande grep peut etre enrichie grace a un certain nombre d’options nteressantes.

vous pouvez indiquer a Git de resumer le resultat en ne montrant que les fichiers et le nombre de correspondances au moyen de l’option --count :

$ git grep --count gmtime\_r

Si vous souhaitez voir dans quelle methode ou fonction la correspondance a ete trouvee, vous pouvez passer l’option -p :

$ git grep -p gmtime\_r \*.c

Vous pouvez aussi rechercher des combinaisons plus complexes de chaines de caracteres avec l’option --and

Ici, nous allons utiliser les options --break et --heading qui aident a decouper le resultat dans un format plus digeste.

Par exemple, recherchons toutes les lignes qui definissent une constante qui contient au choix ≪ LINK ≫ ou ≪ BUF\_MAX ≫ dans la base de code de Git avant la version 1.8.0.

$ git grep --break --heading -n -e '#define' --and ( -e LINK -e BUF\_MAX ) v1.8.0

La commande git grep a quelques avantages sur les commandes de recherche normales telles que grep et ack. Le premier est qu’elle est vraiment rapide, le second est qu’elle vous permet de rechercher dans n’importe quelle arborescence Git, pas seulement la copie de travail. Comme nous l’avons vu dans l’exemple ci-dessus, nous avons cherche des termes dans une version ancienne du code source de Git, pas dans la derniere version extraite.

**Recherche dans le journal Git**

Peut-etre ne cherchez-vous pas **ou** un terme apparait, mais plutot **quand** il existait ou fut introduit. La commande git log comprend un certain nombre d’outils puissants pour trouver des *commits* specifiques par le contenu de leurs messages ou le contenu des diffs qu’ils introduisent.

Si vous voulez trouver par exemple quand la constante ZLIB\_BUF\_MAX a ete initialement introduite, nous pouvons indiquez a Git de ne montrer que les *commits* qui soit ajoutent soit retirent la chaine avec l’option -S.

$ git log -SZLIB\_BUF\_MAX –oneline

Si vous devez etre plus specifique, vous pouvez fournir une expression reguliere a rechercher avec l’option -G.

**Recherche des evolutions d’une ligne**

On l’active avec l’option -L de git log et elle vous montre l’historique d’une fonction ou d’une ligne dans votre base de code.

Par exemple, si nous souhaitions voir toutes les modifications realisees sur la fonction git\_deflate\_bound dans le fichier zlib.c, nous pourrions lancer git log -L

:git\_deflate\_bound:zlib.c. Cette commande va essayer de determiner les limites de cette fonction et de rechercher dans l’historique chaque modification realisee sur la fonction comme une serie de patchs jusqu’au moment de sa creation.

$ git log -L :git\_deflate\_bound:zlib.c

Si Git ne peut pas determiner comment trouver la fonction ou la methode dans votre langage de programmation, vous pouvez aussi fournir une regex. Par exemple, cela aurait donne le meme resultat avec

git log -L '/unsigned long git\_deflate\_bound/',/^}/:zlib.c. Vous auriez pu aussi specifier un intervalle de lignes ou un numero de ligne et vous auriez obtenu le meme type de resultat.

**Reecrire l’historique**

**Modifier la derniere validation**

$ git commit –amend

Vous indexez les modifications que vous voulez en executant git add ou git rm, et le prochain git commit –amend prendra votre index courant et en fera le contenu de votre nouvelle validation.

* **Modifier plusieurs messages de validation**

Par exemple, si vous voulez modifier les 3 derniers messages de validation ou n’importe lequel des messages dans ce groupe, vous fournissez a git rebase -i le parent du dernier *commit* que vous voulez editer,

$ git rebase -i HEAD~3

Souvenez-vous egalement que ceci est une commande de rebasage, chaque *commit* inclus dans l intervalle HEAD~3..HEAD sera reecrit,

* **Reordonner les** *commits*
* **Ecraser un** *commit*
* **Diviser un** *commit*

Pour diviser un *commit*, il doit etre defait, puis partiellement indexe et valide autant de fois que vous voulez pour en finir avec lui

vous pouvez annuler (**reset**) ce *commit* et revalider les modifications que vous voulez pour creer plusieurs *commits*

$ git reset HEAD^

$ git add README

$ git commit -m 'updated README formatting'

$ git add lib/simplegit.rb

$ git commit -m 'added blame'

$ git rebase --continue

**L’option nucleaire :** filter-branch

Il existe une autre option de la reecriture d’historique que vous pouvez utiliser si vous avez besoin de reecrire un grand nombre de *commits* d’une maniere scriptable ; par exemple, modifier globalement votre adresse mail ou supprimer un fichier de tous les *commits*. La commande est filter-branch, et elle peut réécrire des pans entiers de votre historique, vous ne devriez donc pas l’utiliser a moins que votre projet ne soit pas encore public ou que personne n’ait encore travaille

sur les *commits* que vous allez reecrire. Cependant, cela peut etre tres utile. Vous allez maintenant apprendre quelques usages communs pour vous donner une idee de ses capacites

**Supprimer un fichier de chaque** *commit*

Cela arrive assez frequemment. Quelqu’un a accidentellement valide un enorme fichier binaire avec une commande git add . irreflechie, et vous voulez le supprimer partout. Vous avez peut-etre valide un fichier contenant un mot de passe et vous voulez rendre votre projet open source. Filterbranch est l’outil que vous voulez probablement utiliser pour nettoyer votre historique entier. Pour supprimer un fichier nomme ≪ passwords.txt ≫ de tout votre historique, vous pouvez utiliser l’option

--tree-filter de filter-branch :

$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD

Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)

Ref 'refs/heads/master' was rewritten

L’option --tree-filter execute la commande specifiee pour chaque *commit* et le revalide ensuite. Dans le cas present, vous supprimez le fichier nomme ≪ passwords.txt ≫ de chaque contenu, qu’il existait ou non. Si vous voulez supprimer tous les fichiers temporaires des editeurs validesaccidentellement, vous pouvez executer une commande telle que

git filter-branch --tree-filter "find \* -type f -name '\*~' -delete" HEAD.

Vous pourrez alors regarder Git reecrire l’arbre des *commits* et revalider a chaque fois, pour finir en modifiant la reference de la branche. C’est generalement une bonne idee de le faire dans un branche de test puis de faire une reinitialisation forte (**hard-reset**) de votre branche master si le resultat vous convient. Pour executer filter-branch sur toutes vos branches, vous pouvez ajouter --all a la commande.

**Reset demystifie**

Le moyen le plus simple de penser a reset et checkout consiste a representer Git comme un gestionnaire de contenu de trois arborescences differentes.

Git, comme systeme, gere et manipule trois arbres au cours de son operation normale :

**Arbre Role**

HEAD instantane de la derniere validation, prochain parent

Index instantane propose de la prochaine validation

Repertoire de travail bac a sable

**HEAD**

HEAD est un pointeur sur la reference de la branche actuelle, qui est a son tour un pointeur sur le dernier *commit* realise sur cette branche. Ceci signifie que HEAD sera le parent du prochain *commit* a creer.

$ git cat-file -p HEAD

$ git ls-tree -r HEAD

**L’index**

L’index est votre **prochain commit propose**. Nous avons aussi fait reference a ce concept comme la ≪ zone de preparation ≫ de Git du fait que c’est ce que Git examine lorsque vous lancez git commit.

$ git ls-files –s

Encore une fois, nous utilisons ici ls-files qui est plus une commande de coulisses qui vous montre l’etat actuel de votre index. L’index n’est pas techniquement parlant une structure arborescente ‑ c’est en fait un manifeste aplati ‑ mais pour nos besoins, c’est suffisamment proche.

**Le repertoire de travail**

Il faut penser a la copie de travail comme un **bac a sable** ou vous pouvez essayer vos modifications avant de les transferer dans votre index puis le valider dans votre historique.

Reset manipule directement les trois arborescences d’une manière simple et prédictible. Il réalise jusqu’à trois opérations basiques.

**Etape 1: deplacer HEAD**

La premiere chose que reset va faire consiste a deplacer ce qui est pointe par HEAD. Ce n’est pas la meme chose que changer HEAD lui-meme (ce que fait checkout). reset deplace la branche que HEAD pointe.

Quelle que soit la forme du reset que vous invoquez pour un *commit*, ce sera toujours la premiere chose qu’il tentera de faire. Avec reset --soft, il n’ira pas plus loin.

**Etape 2 : Mise a jour de l’index (--mixed)**

**Etape 3: Mise a jour de la copie de travail (--hard)**

La troisième chose que reset va faire est de faire correspondre la copie de travail avec l’index. Si vous utilisez l’option --hard, il continuera avec cette etape.

Il est important de noter que cette option (--hard) est le seul moyen de rendre la commande reset dangereuse et est un des tres rares cas ou Git va reellement detruire de la donnee. Toute autre invocation de reset peut etre defaite,

**Recapitulatif**

La commande reset remplace ces trois arbres dans un ordre specifique, s’arretant lorsque vous lui indiquez :

1. Deplace la branche pointee par HEAD *(s’arrete ici si* --soft*)*

2. Fait ressembler l’index a HEAD *(s’arrete ici a moins que* --hard*)*

3. Fait ressembler le repertoire de travail a l’index.

**Reset avec un chemin**

Donc, supposons que vous lancez git reset file.txt. Cette forme (puisque vous n’avez pas specifie un SHA-1 de commit ni de branche, et que vous n’avez pas non plus specifie --soft ou --hard) est un raccourcis pour git reset --mixed HEAD file.txt.

il ne fait que copier file.txt de HEAD vers index

Ceci a l’effet pratique de *desindexer* le fichier. Si on regarde cette commande dans le diagramme et qu’on pense a ce que git add fait, ce sont des opposes exacts

Nous pourrions tout aussi bien ne pas laisser Git considerer que nous voulions dire ≪ tirer les donnees depuis HEAD ≫ en specifiant un *commit* specifique d’ou tirer ce fichier. Nous lancerions juste quelque chose comme git reset eb43bf file.txt.

**Et checkout**

Comme reset, checkout manipule les trois arborescences et se comporte generalement differemment selon que vous indiquez un chemin vers un fichier ou non.

**Sans chemin**

Lancer git checkout [branche] est assez similaire a lancer git reset --hard [branche] en ce qu’il met a jour les trois arborescences pour qu’elles ressemblent a [branche], mais avec deux differences majeures

Premierement, a la difference de reset --hard, checkout preserve le repertoire de travail ; il s’assure de ne pas casser des fichiers qui ont change. En fait, il est meme un peu plus intelligent que ca – il essaie de faire une fusion simple dans le repertoire de travail, de facon que tous les fichiers **non modifies** soient mis a jour. reset --hard, par contre, va simplement tout remplacer unilateralement sans rien verifier.

Donc, dans les deux cas, nous deplacons HEAD pour pointer sur le commit A, mais la manière differe beaucoup. reset va deplacer la branche pointee par HEAD, alors que checkout va deplacer HEAD lui-meme.



**Avec des chemins**

L’autre facon de lancer checkout est avec un chemin de fichier, ce qui, comme reset, ne deplace pas HEAD. Cela correspond juste a git reset [branche] fichier car cela met a jour l’index avec ce fichier a ce *commit*, mais en remplacant le fichier dans le repertoire de travail. Ce serait exactement comme git reset --hard [branche] fichier (si reset le permettait) – cela ne preserve pas le repertoire de travail et ne deplace pas non plus HEAD.

**Resume**

Voici un aide-memoire sur ce que chaque commande affecte dans chaque arborescence. La colonne ≪ HEAD ≫ contient ≪ REF ≫ si cette commande deplace la reference (branche) pointee par HEAD, et ≪ HEAD ≫ si elle deplace HEAD lui-meme. Faites particulierement attention a la colonne ≪ preserve RT ? ≫ (preserve le repertoire de travail) – si elle indique **NON**, reflechissez a deux fois avant de lancer la commande.



**Fusion avancee**

**Conflits de fusion**

Premierement, nous entrons dans l’etat de conflit de fusion. Puis, nous voulons obtenir des copies de la version locale (**ours**), de la version distante (**theirs**, celle qui vient de la branche a fusionner) et de la version commune (l’ancetre commun depuis lequel les branches sont parties). Ensuite, nous voulons corriger au choix la version locale ou la distante et reessayer de fusionner juste ce fichier.

Obtenir les trois versions des fichiers est en fait assez facile. Git stocke toutes ces versions dans l’index sous formes d’etapes (**stages**), auxquelles chacune y a un nombre associe. Stage 1 est l’ancetre commun, stage 2 est notre version, stage 3 est la version de MERGE\_HEAD, la version qu’on cherche a fusionner (**theirs**).

Vous pouvez extraire une copie de chacune de ces versions du fichier en conflit avec la commande git show et une syntaxe speciale.

$ git show :1:hello.rb > hello.common.rb

$ git show :2:hello.rb > hello.ours.rb

$ git show :3:hello.rb > hello.theirs.rb

Si vous voulez rentrer un peu plus dans le dur, vous pouvez aussi utiliser la commande de plomberie ls-files -u pour recuperer les SHA-1 des blobs Git de chacun de ces fichiers.

$ git ls-files –u

La syntaxe :1:hello.rb est juste un raccourcis pour la recherche du SHA-1 de ce blob.

A present que nous avons le contenu des trois etapes dans notre repertoire de travail, nous pouvons reparer manuellement la copie distante pour resoudre le probleme d’espaces et refusionner le fichier avec la commande meconnue git merge-file dont c’est l’exacte fonction

$ git merge-file -p hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

vous pouvez demander a git diff de comparer le contenu de votre repertoire de travail que vous etes sur le point de valider comme resultat de la fusion avec n’importe quelle etape. Detaillons chaque comparaison. Pour comparer votre resultat avec ce que vous aviez dans votre branche avant la fusion, en d’autres termes, ce que la fusion a introduit, vous pouvez lancer git diff –ours

Si nous voulons voir le resultat de la fusion modifiee depuis la version distante, nous pouvons lancer git diff --theirs.

Enfin, nous pouvons voir comment le fichier a ete modifie dans les deux branches avec git diff --base.

**Examiner les conflits**

Si nous ouvrons le fichier, nous verrons quelque chose comme :

#! /usr/bin/env ruby

def hello

<<<<<<< HEAD

puts 'hola world'

======

puts 'hello mundo'

>>>>>>> mundo

end

hello()

Peut-etre le moyen de resoudre n’est-il pas evident. Il necessite plus de contexte.

Un outil utile est git checkout avec l’option --conflict. Il va re-extraire le fichier et remplacer les marqueurs de conflit. Cela peut etre utile si vous souhaitez eliminer les marqueurs et essayer de resoudre le conflit a nouveau.

Vous pouvez passer en parametre a --conflict, soit diff3 soit merge (le parametre par defaut). Si vous lui passez diff3, Git utilisera une version differente des marqueurs de conflit, vous fournissant non seulement les versions locales (*ours*) et distantes (*theirs*), mais aussi la version ≪ base ≫ integree pour vous fournir plus de contexte.

$ git checkout --conflict=diff3 hello.rb

Une fois que nous l’avons lance, le fichier ressemble a ceci :

#! /usr/bin/env ruby

def hello

<<<<<<< ours

puts 'hola world'

||||||| base

puts 'hello world'

======

puts 'hello mundo'

>>>>>>> theirs

end

hello()

Si vous appreciez ce format, vous pouvez le regler comme defaut pour les futur conflits de fusion en renseignant le parametre merge.conflictstyle avec diff3.

$ git config --global merge.conflictstyle diff3

La commande git checkout peut aussi accepter les options --ours et --theirs, qui peuvent servir de moyen rapide de choisir unilateralement une version ou une autre sans fusion

Cela peut etre particulierement utile pour les conflits de fichiers binaires ou vous ne pouvez que choisir un des cote, ou des conflits ou vous souhaitez fusionner certains fichiers depuis d’autres branches - vous pouvez fusionner, puis extraire certains fichiers depuis un cote ou un autre avant de valider le resultat.

**Journal de fusion**

Un autre outil utile pour la resolution de conflits de fusion est git log. Cela peut vous aider a obtenir du contexte ce qui a contribue aux conflits. Parcourir un petit morceau de l’historique pour se rappeler pourquoi deux lignes de developpement ont touche au meme endroit dans le code peut s’averer quelque fois tres utile.

Pour obtenir une liste complete de tous les *commits* uniques qui ont ete introduits dans chaque branche impliquee dans la fusion, nous pouvons utiliser la syntaxe ≪ triple point ≫ que nous avons apprise dans Triple point.

$ git log --oneline --left-right HEAD...MERGE\_HEAD

Neanmoins, nous pouvons simplifier encore plus ceci pour fournir beaucoup plus de contexte. Si nous ajoutons l’option --merge a git log, il n’affichera que les *commits* de part et d’autre de la fusion qui modifient un fichier presentant un conflit.

$ git log --oneline --left-right –merge

Si nous lancons cela avec l’option -p a la place, vous obtenez les diffs limites au fichier qui s’est retrouve en conflit. Cela peut s’averer **vraiment** utile pour vous donner le contexte necessaire a la comprehension de la raison d’un conflit et a sa resolution intelligente.

**Defaire des fusions**

**Correction des references**

git reset --hard HEAD~

Le defaut de cette approche est qu’elle re-ecrit l’historique, ce qui peut etre problematique avec un depot partage.

**Inverser le** *commit*

Si les deplacements des pointeurs de branche ne sont pas envisageables, Git vous donne encore l’option de creer un nouveau *commit* qui defait toutes les modifications d’un autre deja existant. Git appelle cette option une ≪ inversion ≫ (*revert*), et dans ce scenario particulier, vous l’invoqueriez comme ceci :

$ git revert -m 1 HEAD

**Autres types de fusions**

**Preference** *our* **ou** *theirs*

Premierement, il existe un autre mode utile que nous pouvons utiliser avec le mode ≪ recursive ≫ normal de fusion. Nous avons deja vu les options ignore-all-space et ignore-space-change qui sont passees avec -X mais nous pouvons aussi indiquer a Git de favoriser un cote plutot que l’autre lorsqu’il rencontre un conflit.

Si vous preferez que Git choisisse simplement un cote specifique et qu’il ignore l’autre cote au lieu de vous laisser fusionner manuellement le conflit, vous pouvez passer -Xours ou -Xtheirs a la commande merge.

Si une des options est specifiee, Git ne va pas ajouter de marqueurs de conflit. Toutes les differences qui peuvent etre fusionnees seront fusionnees. Pour toutes les differences qui generent un conflit, Git choisira simplement la version du cote que vous avez specifie, y compris pour les fichiers binaires.

$ git merge -Xours mundo

Cette option peut aussi etre passee a la commande git merge-file que nous avons utilisee plus tot en lancant quelque chose comme git merge-file --ours pour les fusions de fichiers individuels

Si vous voulez faire quelque chose similaire mais indiquer a Git de ne meme pas essayer de fusionner les modifications de l’autre cote, il existe une option draconienne qui s’appelle la *strategie* de fusion ≪ *ours* ≫.

Cela realisera une fusion factice. Cela enregistrera un nouveau *commit* de fusion avec les deux branches comme parents, mais ne regardera meme pas la branche en cours de fusion. Cela enregistrera simplement le code exact de la branche courante comme resultat de la fusion.

$ git merge -s ours mundo

Vous pouvez voir qu’il n’y a pas de difference entre la branche sur laquelle nous etions precedemment et le resultat de la fusion.

**Deboguer avec Git**

vous pouvez visualiser le fichier annote avec git blame pour voir quand chaque ligne de la methode a ete modifiee pour la derniere fois et par qui. Cet exemple utilise l’option -L pour limiter la sortie des lignes 12 a 22 :

$ git blame -L 12,22 simplegit.rb

**Sous-modules**

**Demarrer un sous-module**

Pour ajouter un nouveau sous-module, nous utilisons la commande git submodule add avec l’URL du projet que nous souhaitons suivre

$ git submodule add https://github.com/chaconinc/DbConnector

Si vous lancez git status a ce moment, vous noterez quelques differences.

Premierement, un fichier .gitmodules vient d’apparaitre. C’est le fichier de configuration qui stocke la liaison entre l’URL du projet et le sous-repertoire local dans lequel vous l’avez tire.

$ cat .gitmodules

[submodule "DbConnector"]

path = DbConnector

url = <https://github.com/chaconinc/DbConnector>

fichier. Il est important de noter que ce fichier est en gestion de version comme vos autres fichiers, a l’instar de votre fichier .gitignore. Il est pousse et tire comme le reste de votre projet.

$ git diff DbConnector

Meme si DbConnector est un sous-repertoire de votre repertoire de travail, Git le voit comme un sous-module et ne suit pas son contenu (si vous n’etes pas dans ce repertoire). En echange, Git l’enregistre comme un *commit* particulier de ce depot. Si vous souhaitez une sortie diff plus agreable, vous pouvez passer l’option --submodule a git diff.

$ git diff –submodule

**Cloner un projet avec des sous-modules**

Maintenant, vous allez apprendre a cloner un projet contenant des sous-modules. Quand vous recuperez un tel projet, vous obtenez les differents repertoires qui contiennent les sous-modules, mais encore aucun des fichiers

Le repertoire DbConnector est present mais vide. Vous devez executer deux commandes : git submodule init pour initialiser votre fichier local de configuration, et git submodule update pour tirer toutes les donnees de ce projet et recuperer le *commit* approprie tel que liste dans votre superprojet:

$ git submodule init

$ git submodule update

Il existe une autre maniere plus simple d’arriver au meme resultat. Si vous passez l’option --recurse-submodules a la commande git clone, celle-ci initialisera et mettra a jour automatiquement chaque sous-module du depot.

**Travailler sur un projet comprenant des sous-modules**

**Tirer des modifications amont**

Quand vous souhaitez vérifier si le sous-module a évolue, vous pouvez vous rendre dans le répertoire correspondant et lancer git fetch puis git merge de la branche amont pour mettre a jour votre code local.

$ git fetch

$ git merge origin/master

Si vous revenez maintenant dans le projet principal et lancez git diff --submodule, vous pouvez remarquer que le sous-module a ete mis a jour et vous pouvez obtenir une liste des *commits* qui y ont ete ajoutes

Il existe aussi un moyen plus facile, si vous preferez ne pas avoir a recuperer et fusionner manuellement les modifications dans le sous-repertoire. Si vous lancez la commande git submodule update --remote, Git se rendra dans vos sous-modules et realisera automatiquement le fetch et le merge.

C’est une information interessante car vous pouvez voir le journal des modifications que vous vous appretez a valider dans votre sous-module. Une fois validees, vous pouvez encore visualiser cette information en lancant git log -p.

$ git log -p –submodule

Par defaut, Git essaiera de mettre a jour **tous** les sous-modules lors d’une commande git submodule update --remote, donc si vous avez de nombreux sous-modules, il est preferable de specifier le sousmodule que vous souhaitez mettre a jour.

$ git submodule update --remote DbConnector

**Travailler sur un sous-module**

En premier, rendons-nous dans le repertoire de notre sous-module et extrayons une branche.

$ git checkout stable

Attaquons-nous au choix de politique de gestion. Pour le specifier manuellement, nous pouvons simplement ajouter l’option --merge or --rebasea l’appel de update.

$ git submodule update --remote –merge

Maintenant, si nous mettons a jour notre sous-module, nous pouvons voir ce qui arrive lors d’un rebasage de deux modifications concurrentes.

$ git submodule update --remote –rebase

**Publier les modifications dans un sous-module**

Pour etre sur que cela n’arrive pas, vous pouvez demander a Git de verifier que tous vos sousmodules ont ete correctement pousses avant de pouvoir pousser le projet principal. La commande git push accepte un argument --recurse-submodules qui peut avoir pour valeur ≪ *check* ≫ ou ≪ *on-demand* ≫. L’option ≪ check ≫ fera echouer push si au moins une des modifications des sous-modules n’a pas ete poussee.

Comme vous pouvez le voir, Git s’est rendu dans le module DbConnector et l’a pousse avant de pousser le projet principal. Si la poussee du sous-module echoue pour une raison quelconque, la poussee du projet principal sera annulee.

**Trucs et astuces pour les sous-modules**

Il existe quelques commandes qui permettent de travailler plus facilement avec les sous-modules.

**Submodule foreach**

Il existe une commande submodule foreach qui permet de lancer une commande arbitraire dans chaque sous-module. C’est particulierement utile si vous avez plusieurs sous-modules dans le meme projet

$ git submodule foreach git stash

Ensuite, nous pouvons creer une nouvelle branche et y basculer dans tous nos sous-modules.

$ git submodule foreach 'git checkout -b featureA'

$ git diff; git submodule foreach 'git diff'

Les alias Git, mais voici un exemple d’alias que vous pourriez trouver utiles si vous voulez travailler serieusement avec les sous-modules de Git.

$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff'"

$ git config alias.spush 'push --recurse-submodules=on-demand'

$ git config alias.supdate 'submodule update --remote --merge'

**Configuration de Git**

**Configuration de base d’un client**

core.editor:

Pour modifier ce programme par defaut pour un autre, vous pouvez utiliser le parametre core.editor

$ git config --global core.editor emacs

commit.template:

Si vous reglez ceci sur le chemin d’un fichier sur votre systeme, Git utilisera ce fichier comme message par defaut quand vous validez

$ git config --global commit.template ~/.gitmessage.txt

core.autocrlf

$ git config --global core.autocrlf true

$ git config --global core.autocrlf input

Si vous etes un programmeur Windows gerant un projet specifique a Windows, vous pouvez desactiver cette fonctionnalite et forcer l’enregistrement des ≪ retour chariot ≫ dans le depot en reglant la valeur du parametre a false :

$ git config --global core.autocrlf false

**Plomberie et porcelaine**

Quand vous executez git init dans un nouveau repertoire ou un repertoire existant, Git cree un repertoire .git qui contient presque tout ce que Git stocke et manipule.

$ ll -la .git

total 11

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 ./

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 ../

-rw-r--r-- 1 asaki 1049089 130 Sep 27 18:37 config

-rw-r--r-- 1 asaki 1049089 73 Sep 27 18:37 description

-rw-r--r-- 1 asaki 1049089 23 Sep 27 18:37 HEAD

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 hooks/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 info/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 objects/

drwxr-xr-x 1 asaki 1049089 0 Sep 27 18:37 refs/

Vous y verrez sans doute d’autres fichiers, mais ceci est un depot qui vient d’etre cree avec git init et c’est ce que vous verrez par defaut. Le fichier description est utilise uniquement par le programme GitWeb, il ne faut donc pas s’en soucier. Le fichier config contient les options de configuration specifiques a votre projet et le repertoire info contient un fichier d’exclusions listant les motifs que vous souhaitez ignorer et que vous ne voulez pas mettre dans un fichier .gitignore. Le repertoire hooks contient les scripts de procedures automatiques cote client ou serveur, ils sont decrits en detail dans Crochets Git.

Il reste quatre elements importants : les fichiers HEAD et (pas encore cree) index, ainsi que les repertoires objects et refs. Ce sont les composants principaux d’un depot Git. Le repertoire objects stocke le contenu de votre base de donnees, le repertoire refs stocke les pointeurs vers les objets *commit* de ces donnees (branches), le fichier HEAD pointe sur la branche qui est en cours dans votre repertoire de travail et le fichier index est l’endroit ou Git stocke les informations sur la zone d’attente. Vous allez maintenant plonger en detail dans chacune de ces sections et voir comment Git fonctionne.

**Les objets de Git**

Git est un systeme de fichier adressables par contenu. Ca veut dire que le coeur de Git est une simple base de paires cle/valeur. Vous pouvez y inserer n’importe quelle sorte de donnees et il vous retournera une cle que vous pourrez utiliser a n’importe quel moment pour recuperer ces donnees. Pour illustrer cela, vous pouvez utiliser la commande de plomberie hash-object, qui prend des donnees, les stocke dans votre repertoire .git, puis retourne la cle sous laquelle les donnees sont stockees

$ find .git/objects

.git/objects

.git/objects/info

.git/objects/pack

$ find .git/objects -type f

Git a initialise le repertoire objects et y a cree les sous-repertoires pack et info, mais ils ne contiennent pas de fichier regulier.

$ echo 'test content' | git hash-object -w –stdin

L’option -w specifie a hash-object de stocker l’objet, sinon la commande repondrait seulement quelle serait la cle. --stdin specifie a la commande de lire le contenu depuis l’entree standard, sinon hash-object s’attend a trouver un chemin vers un fichier. La sortie de la commande est une empreinte de 40 caracteres. C’est l’empreinte SHA-1 ‒ une somme de controle du contenu du fichier que vous stockez plus un en-tete,

Voyez maintenant comment Git a stocke vos donnees :

$ find .git/objects -type f

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

C’est comme cela que Git stocke initialement du contenu ‒ un fichier par contenu, nomme d’apres la somme de controle SHA-1 du contenu et de son en-tete. Le sous-repertoire est nomme d’apres les 2 premiers caracteres de l’empreinte et le

fichier d’apres les 38 caracteres restants.

Vous pouvez recuperer le contenu avec la commande cat-file. Cette commande est un peu le couteau suisse pour l’inspection des objets Git. Lui passer l’option -p ordonne a la commande catfile de determiner le type de contenu et de vous l’afficher joliment :

$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4

D’abord, creez un nouveau fichier et enregistrez son contenu dans la base de donnees:

$ echo 'version 1' > test.txt

$ git hash-object -w test.txt

83baae61804e65cc73a7201a7252750c76066a30

Puis, modifiez le contenu du fichier et enregistrez-le a nouveau :

$ echo 'version 2' > test.txt

$ git hash-object -w test.txt

1f7a7a472abf3dd9643fd615f6da379c4acb3e3a

Votre base de donnees contient les 2 versions du fichier, ainsi que le premier contenu que vous avez stocke ici :

$ find .git/objects -type f

.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a

.git/objects/83/baae61804e65cc73a7201a7252750c76066a30

.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4

Vous pouvez maintenant restaurer le fichier a sa premiere version :

$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt

$ cat test.txt

version 1

Git peut vous donner le type d’objet de n’importe quel objet Git, etant donne sa cle SHA-1, avec cat-file -t :

**Les objets arbres**

Un unique arbre contient une ou

plusieurs entrees, chacune etant l’empreinte SHA-1 d’un blob ou d’un sous-arbre (*sub-tree*) avec ses droits d’acces (*mode*), son type et son nom de fichier associes. L’arbre le plus recent d’un projet pourrait ressembler, par exemple, a ceci :

$ git cat-file -p master^{tree}

100644 blob a906cb2a4a904a152e80877d4088654daad0c859 README

100644 blob 8f94139338f9404f26296befa88755fc2598c289 Rakefile

040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0 lib

La syntaxe master^{tree} signifie l’objet arbre qui est pointe par le dernier *commit* de la branche master

Ces trois objets principaux (le blob, l’arbre et le *commit*) sont initialement stockes dans des fichiers separes du repertoire .git/objects. Voici tous les objets

contenus dans le repertoire exemple, commentes d’apres leur contenu :

On trouve les fichiers contenant des empreintes SHA-1 dans le repertoire git/refs.

$ find .git/refs

.git/refs

.git/refs/heads

.git/refs/tags

$ find .git/refs -type f

C’est simplement ce qu’est une branche dans Git : un simple pointeur ou reference sur le dernier etat d’une suite de travaux. Pour creer une branche a partir du deuxieme *commit*, vous pouvez faire ceci :

**La branche HEAD**

Le fichier HEAD est une reference symbolique a la branche courante. Par reference symbolique, j’entends que contrairement a une reference normale, elle ne contient pas une empreinte SHA-1, mais plutot un pointeur vers une autre reference. Si vous regardez ce fichier, vous devriez voir quelque chose comme ceci :

$ cat .git/HEAD

ref: refs/heads/master

**Etiquettes**

Nous venons de parcourir les trois types d’objets utilises par Git, mais il en existe un quatrieme. L’objet etiquette (*tag* en anglais) ressemble beaucoup a un objet *commit*. Il contient un etiqueteur, une date, un message et un pointeur. La principale difference est que l’etiquette pointe en general vers un *commit* plutot qu’un arbre. C’est comme une reference a une branche, mais elle ne bouge jamais : elle pointe toujours vers le meme *commit*, lui donnant un nom plus sympathique.

Comme presente au Les bases de Git, il existe deux types d’etiquettes : annotee et legere

C’est tout ce qu’est une etiquette legere : une reference qui n’est jamais modifiee.

Une etiquette annotee est plus complexe. Quand on cree une etiquette annotee, Git cree un objet etiquette, puis enregistre une reference qui pointe vers lui plutot que directement vers le *commit*. Vous pouvez voir ceci en creant une etiquette annotee (-a specifie que c’est une etiquette annotee) :

$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'

Voici l’empreinte SHA-1 de l’objet cree :

$ cat .git/refs/tags/v1.1

9585191f37f7b0fb9444f35a9bf50de191beadc2

**References distantes**

Git stocke la valeur que vous avez poussee en dernier vers cette reference pour chaque branche dans le repertoire refs/remotes

Ensuite, vous pouvez voir l’etat de la branche master dans la reference distante origin la derniere fois que vous avez communique avec le serveur en regardant le fichier refs/remotes/origin/master :

$ cat .git/refs/remotes/origin/master

ca82a6dff817ec66f44342007202690a93763949

**Recuperation de donnees**

A chaque *commit* ou commutation de branche, le journal des references (*reflog*) est mis a jour

Vous pouvez voir ou vous etiez a n’importe quel moment en executant git reflog . Pour voir, les memes informations d’une maniere plus utile, nous pouvons executer git log -g, qui nous donnera une sortie normalisee pour votre journal de references

Comme les donnees du journal de reference sont sauvegardees dans le repertoire .git/logs/, vous n’avez effectivement plus de journal de references. Comment pouvez-vous recuperer ces *commits* maintenant ? Une maniere de faire est d’utiliser l’outil git fsck, qui verifie l’integrite de votre base de donnees. Si vous l’executez avec l’option --full, il vous montre tous les objets qui ne sont pas

references par d’autres objets :

$ git fsck –full

Vous pouvez executer la commande count-objects pour voir rapidement combien d’espace disque vous utilisez :

$ git count-objects -v

**Github**

git ls-remote origin affcihent la liste de toutes les branches et etiquettes ainsi que d’autres references dans le depot.

**Git dans d’autres environnements**

gitk est l’outil de visualisation graphique d’historique. Voyez-le comme une interface GUI puissante par-dessus git log et git grep. C’est l’outil a utiliser lorsque vous essayez de trouver un evenement passe ou de visualiser l’historique de votre projet.

Gitk accepte de nombreuses options de ligne de commande, dont la plupart sont passees directement a la commande git log sous-jacente. L’une des plus interessantes est probablement d’ajouter l’option --all qui indique a gitk de montrer tous les *commits* joignables depuis *n’importe quelle* reference, et pas seulement HEAD.

git-gui, par contre est un outil permettant de ciseler les *commits*. Lui aussi est plus facile a invoquer en ligne de commande :

$ git gui

**Commandes Git**

**git config**

**git clone**

La commande git clone sert en fait a englober plusieurs autres commandes. Elle cree un nouveau dossier, va a l’interieur de celui-ci et lance git init pour en faire un depot Git vide, ajoute un serveur distant (git remote add) a l’URL que vous lui avez passee (appele par defaut origin), lance git fetch a partir de ce depot distant et ensuite extrait le dernier *commit* dans votre repertoire de travail avec git checkout.

Enfin, dans Cloner un projet avec des sous-modules, nous apprenons l’option --recursive-submodule pour rendre le clonage d’un depot avec sous-modules un peu plus simple.

**git diff**

La commande git diff s’utilise lorsque vous voulez voir la difference entre deux arbres. Cela peut etre la difference entre votre repertoire de travail et votre index (git diff en elle-meme), entre votre index et votre dernier *commit* (git diff --staged) ou entre deux *commits* (git diff master brancheB).

Nous l’utilisons pour chercher de possibles problemes d’espaces blancs avant de valider avec l’option --check dans Guides pour une validation.

Nous voyons comment verifier les differences entre branches plus efficacement avec la syntaxe git diff A...B dans Determiner les modifications introduites.

Nous l’utilisons pour filtrer les differences d’espaces blancs avec -w et comment comparer differentes etapes de fichiers conflictuels avec --theirs, --ours et --base dans Fusion avancee. Enfin, nous l’utilisons pour comparer efficacement les modifications de sous-modules avec --submodule dans Demarrer un sous-module.

**git reset**

La commande git reset est d’abord utilisee pour defaire des choses, comme son nom l’indique. Elle modifie le pointeur HEAD et change optionnellement l’index et peut aussi modifier le repertoire de travail si vous l’utilisez avec l’option --hard. Cette derniere option rend possible la perte de votre travail par cette commande si elle est mal employee, alors soyez certain de bien la comprendre avant de l’utiliser.

**git rm**

Nous couvrons la commande git rm dans le detail dans Effacer des fichiers, y compris en supprimant recursivement les fichiers et en ne supprimant les fichiers que de l’index mais en les laissant dans le repertoire de travail avec --cached.

**git branch**

Dans Suivre les branches, nous utilisons l’option git branch -u pour definir une branche de suivi

**git checkout**

Nous voyons comment l’utiliser pour commencer a suivre des branches avec l’option --track dans Suivre les branches

**git log**

La commande git log est utilisee pour montrer l’historique enregistre atteignable d’un projet en partant du *commit* le plus recent.

La nous regardons les options -p et --stat pour avoir une idee de ce qui a ete introduit dans chaque *commit* et les options --pretty et --oneline pour voir l’historique de maniere plus concise, avec quelques options simples de filtre de date et d’auteur

Dans Creer une nouvelle branche, nous l’utilisons avec l’option --decorate pour visualiser facilement ou se trouvent nos pointeurs de branche et nous utilisons aussi l’option --graph pour voir a quoi ressemblent les historiques divergents.

Dans Cas d’une petite equipe privee et Plages de *commits*, nous couvrons la syntaxe brancheA..brancheB que nous utilisons avec la commande git log pour voir quels *commits* sont propres a une branche relativement a une autre branche. Dans Plages de *commits*, nous explorons cela de maniere assez detaillee

Dans Journal de fusion et Triple point, nous couvrons l’utilisation du format brancheA...brancheB et de la syntaxe --left-right pour voir ce qui est dans une branche ou l’autre mais pas dans les deux a la fois. Dans Journal de fusion, nous voyons aussi comment utiliser l’option --merge comme aide au debogage de conflit de fusion tout comme l’utilisation de l’option --cc pour regarder les conflits

de *commits* de fusion dans votre historique.

Dans Raccourcis RefLog, nous utilisons l’option -g pour voir le reflog Git a travers cet outil au lieu de faire le parcours de la branche.

Dans Recherche, nous voyons l’utilisation des options -S et -L pour faire des recherches assez sophistiquees sur quelque chose qui s’est passe historiquement dans le code comme voir l’historique d’une fonction.

**git stash**

La commande git stash est utilisee pour remiser temporairement du travail non valide afin d’obtenir un repertoire de travail propre sans avoir a valider du travail non termine dans une branche.

**git push**

Nous voyons dans Partager les etiquettes comment s’en servir avec l’option --tags pour partager des etiquettes que vous avez faites.

Dans Publier les modifications dans un sous-module, nous utilisons l’option --recurse-submodules pour verifier que tout le travail de nos sous-modules a ete publie avant de pousser le super-projet, ce qui peut etre vraiment utile quand on utilise des sous-modules

**git remote**

Elle est aussi utilisee dans presque tous les chapitres suivants du livre, mais toujours dans le format standard git remote add <nom> <URL>.

**git submodule**

La commande git submodule est utilisee pour gerer des depots externes a l’interieur de depots normaux. Cela peut etre pour des bibliotheques ou d’autres types de ressources partagees. La commande submodule a plusieurs sous-commandes (add, update, sync, etc) pour la gestion de ces ressources

**Inspection et comparaison**

**git show**

La commande git show peut montrer un objet Git d’une facon simple et lisible pour un etre humain. Vous l’utiliseriez normalement pour montrer les informations d’une etiquette ou d’un *commit*. Nous l’utilisons d’abord pour afficher l’information d’une etiquette annotee dans Les etiquettes annotees.

**git shortlog**

La commande git shortlog est utilisee pour resumer la sortie de git log. Elle prendra beaucoup des memes options que la commande git log mais au lieu de lister tous les *commits*, elle presentera un resume des *commits* groupes par auteur

**git describe**

La commande git describe est utilisee pour prendre n’importe quelle chose qui se resoud en un *commit* et produit une chaine de caracteres qui est somme toute lisible pour un etre humain et qui ne changera pas. C’est une facon d’obtenir une description d’un *commit* qui est aussi claire qu’un SHA de *commit* mais en plus comprehensible.

**Debogage**

**git bisect**

L’outil git bisect est un outil de debogage incroyablement utile utilise pour trouver quel *commit* specifique a le premier introduit un bug ou probleme en faisant une recherche automatique pardichotomie.

**git blame:**

La commande git blame annote les lignes de n’importe quel fichier avec quel *commit* a ete le dernier a introduire un changement pour chaque ligne du fichier et quelle personne est l’auteur de ce *commit*. C’est utile pour trouver la personne pour lui demander plus d’informations sur une section specifique de votre code

**git grep**

La commande git grep peut aider a trouver n’importe quelle chaine de caracteres ou expression reguliere dans n’importe quel fichier de votre code source, meme dans des anciennes versions de votre projet.

**Patchs**

**git cherry-pick**

**git rebase**

La commande git rebase est simplement un cherry-pick automatise. Elle determine une serie de *commits* puis les selectionne et les re-applique un par un dans le meme ordre ailleurs.

**git revert**

La commande git revert est fondamentalement le contraire de git cherry-pick. Elle cree un *commit* qui applique l’exact oppose des modifications introduites par le *commit* que vous ciblez,essentiellement en le defaisant ou en revenant dessus

**Administration**

**git gc**

**git fsck**

La commande git fsck est utilisee pour verifier les problemes ou les incoherences de la base de donnees interne.

**git reflog**

La commande git reflog explore un journal de la ou toutes vos branches sont passees pendant que vous travailliez pour trouver des *commits* que vous pourriez avoir perdus en re-ecrivant des historiques.

git log -g pour visualiser la meme information avec la sortie de git log.

**Commandes de plomberie**

ls-remote pour regarder les references brutes sur le serveur

ls-files

rev-parse

**Git distribue**

$ git checkout -b fonctionBv2 origin/master

$ git merge --no-commit --squash fonctionB

# (changement d'implementation)

$ git commit

$ git push macopie fonctionBv2

L’option --squash prend tout le travail de la branche a fusionner et le colle dans un *commit* sans fusion au sommet de la branche extraite. L’option --no-commit indique a Git de ne pas enregistrer automatiquement une validation. Cela permet de reporter toutes les modifications d’une autre branche, puis de realiser d’autres modifications avant de realiser une nouvelle validation

**Determiner les modifications introduites**

Faire une revue de tous les *commits* dans cette branche s’avere souvent d’une grande aide. Vous pouvez exclure les *commits* de la branche master en ajoutant l’option --not devant le nom de la branche. C’est equivalent au format master..contrib utilise plus haut. Par exemple, si votre contributeur vous envoie deux patchs et que vous creez une branche appelee contrib et y appliquez ces patchs, vous pouvez lancer ceci :

$ git log contrib --not master

Pour visualiser les modifications que chaque **commit** introduit, souvenez-vous que vous pouvez passer l’option -p a git log et elle ajoutera le diff introduit a chaque *commit*.

Pour visualiser un diff complet de ce qui arriverait si vous fusionniez cette branche thematique avec une autre branche, vous pouvez utiliser un truc bizarre pour obtenir les resultats corrects. Vous pourriez penser a lancer ceci :

$ git diff master

Si master est un ancetre directe de la branche thematique, ce n’est pas un probleme. Si les deux historiques ont diverge, le diff donnera l’impression que vous ajoutez toutes les nouveautes de la branche thematique et retirez tout ce qui a ete fait depuis dans la branche master.

Ce que vous souhaitez voir en fait, ce sont les modifications ajoutees sur la branche thematique — le travail que vous introduirez si vous fusionnez cette branche dans master. Vous obtenez ce resultat en demandant a Git de comparer le dernier instantane de la branche thematique avec son ancetre commun a la branche master le plus recent.

Techniquement, c’est realisable en determinant exactement l’ancetre commun et en lancant lacommande diff dessus :

$ git merge-base contrib master

36c7dba2c95e6bbb78df

$ git diff 36c7db

Neanmoins, comme ce n’est pas tres commode, Git fournit un raccourci pour realiser la meme chose : la syntaxe a trois points. Dans le contexte de la commande diff, vous pouvez placer trois points apres une autre branche pour realiser un diff entre le dernier instantane de la branche sur laquelle vous vous trouvez et son ancetre commun avec une autre branche :

$ git diff master...contrib

Cette commande ne vous montre que les modifications que votre branche thematique a introduites depuis son ancetre commun avec master. C’est une syntaxe tres simple a retenir

**Rerere**

Si vous fusionnez et rebasez beaucoup ou si vous maintenez une branche au long cours, la fonctionnalite appelee ≪ rerere ≫ peut s’averer utile.

Rerere signifie ≪ *re* utiliser les *re* solutions en *re* gistrees ≫ (“ *reuse recorded resolution* ”) ‑ c’est un moyen de raccourcir les resolutions manuelles de conflit. Quand rerere est actif, Git va conserver un jeu de couples d’images pre et post fusion des fichiers ayant presente des conflits, puis s’il s’apercoit qu’un conflit ressemble a une de ces resolutions, il va utiliser la meme strategie sans rien

vous demander.

**Generation d’un nom de revision**

Comme Git ne fournit pas par nature de nombres croissants tels que ≪ r123 ≫ a chaque validation, la commande git describe permet de generer un nom humainement lisible pour chaque *commit*. Git concatene le nom de l’etiquette la plus proche, le nombre de validations depuis cette etiquette et un code SHA-1 partiel du *commit* que l’on cherche a definir :

$ git describe master

v1.6.2-rc1-20-g8c5b85c

La commande git describe repose sur les etiquettes annotees (etiquettes creees avec les options –a ou -s). Les etiquettes de publication doivent donc etre creees de cette maniere si vous souhaitez utiliser git describe pour garantir que les *commits* seront decrits correctement. Vous pouvez aussi utiliser ces noms comme cible lors d’une extraction ou d’une commande show, bien qu’ils reposent sur le SHA-1 abrege et pourraient ne pas rester valides indefiniment.

**Preparation d’une publication**

Une des etapes consiste a creer une archive du dernier instantane de votre code pour les malheureux qui n’utilisent pas Git. La commande dediee a cette action est git archive :

$ git archive master --prefix='projet/' | gzip > `git describe master`.tar.gz

$ ls \*.tar.gz

v1.6.2-rc1-20-g8c5b85c.tar.gz

$ git archive master --prefix='project/' --format=zip > `git describe master`.zip

**Shortlog**

Il est temps d’envoyer une annonce a la liste de diffusion des nouveautes de votre projet. Une maniere simple d’obtenir rapidement une sorte de liste des modifications depuis votre derniere version ou courriel est d’utiliser la commande git shortlog. Elle resume toutes les validations dans l’intervalle que vous lui specifiez. Par exemple, ce qui suit vous donne un resume de toutes les validations depuis votre derniere version si celle-ci se nomme v1.0.1 :

$ git shortlog --no-merges master --not v1.0.1